SP-104 Active Learning System Images

Final Report

CS 4850 – Fall 2025

Team Members

Josh Smith
Team Leader
Developer

Elijah Merrill
Developer

Noah Lane
Documentation

Matthew Hall
Documentation

Advisor

Sharon Perry

December 1st, 2025

Link to GitHub Repository: https://github.com/ksusp104/ChestXray_ImageClassifier

Link to Project Website: https://ksusp104.github.io/

Stats and Status

Lines of Code	436
Components/Tools	10
Estimate Hours	360
Actual Hours	453
Status	Project is 85% complete and working as designed

Table of Contents

Introduction

The purpose of this project was to design and develop an active learning system that works with a publicly available chest X-ray dataset. The system focuses on automating the process of selecting and labeling medical images in a way that improves model performance while reducing the amount of manual labeling required. Active learning is useful in medical imaging because it allows a model to learn efficiently from a small number of high-value samples, which is especially important when dealing with large datasets or limited expert involvement.

For this project, the team used a dataset from Kaggle that contains thousands of chest X-ray images along with metadata describing diseases and conditions. The system operates through a Python-based workflow that loads the dataset, builds index files, trains a model, and uses active learning techniques to select additional samples for training. All interaction with the system takes place through the command line, making the setup lightweight and accessible.

The goal of the project was not to create a full clinical tool, but rather to demonstrate the core ideas of active learning and how they can be applied to medical imaging. The system provides a foundation for understanding dataset handling, active learning training, and iterative sample selection. Through this project, the team gained experience with real-world medical imaging data, active learning frameworks, and managing a complete development workflow from setup to testing.

Requirements

Constraints

The system must operate using the provided chest X-ray dataset, which must be downloaded and stored locally by the user before running the program. The program must run on a computer with a modern operating system and a working Python 3 environment. All required Python packages, including PyTorch, pandas, numpy, and others, must be installed correctly for the system to function. File paths must match the user's local directory structure, meaning the dataset and source files must be stored in locations that the program can access. Since the system runs entirely through the command line, it requires no graphical interface. Hardware limitations, such as a lack of processing power or a missing GPU, may slow down training but will not stop the system from running.

Functional Requirements

- The system must be able to load and read the chest X-ray dataset and its associated metadata files.
- The system must generate index files (such as an index table for CSV files) that summarize the dataset and confirm a successful connection.
- The program must support training a machine learning model (PyTorch Libraries).
- The program must support selecting samples through active learning.

- The system must be able to update the model with newly labeled data.
- The system must provide command-line output indicating progress, errors, and results, so users can track each step.
- The system must allow users to run evaluation tasks to measure the accuracy and performance of the trained model.

Non-Functional Requirements

- The system should be dependable and produce consistent results across different computers with the same setup.
- The system should provide clear and readable command-line messages, so users understand what is happening during training or evaluation.
- The system should run efficiently on standard computer hardware, with improved performance on higher-end machines.
- All files created by the program (such as outputs and index tables) should follow a predictable and well-organized structure.
- The system should be easy to set up using the provided instructions.
- The system should not require any advanced technical background to operate.

Assumptions

1. Assumed that the NIH Chest X-Rays dataset from Kaggle will remain publicly available and accessible for the duration of development
2. It is assumed that users of the system will have access to a machine capable of running the program as a web-based tool, interactive notebook, or downloadable application.
3. Assumed that users will have access to an internet connection for downloading the dataset and installing any external dependencies.
4. Assumed that the hosting environment (given if it's deployed as a web-based tool) will support the necessary frameworks and allow secure storage of temporary data.

Analysis / Design

Assumptions and Dependencies

The system will heavily rely on a given dataset to function properly; however, beyond this, there are relatively few dependencies. The dataset serves as the foundation for training, validating, and testing the active learning model, and its quality directly affects the system’s accuracy and reliability. Without sufficient data diversity and size, the system may not generalize well to real-world cases. Otherwise, the system will mainly depend on the computing environment in which it is deployed. It will require reasonably up-to-date hardware, which will be discussed in the following section, to ensure smooth image processing and performance. Additionally, it is assumed that the user will operate the system on a modern operating system, as this will allow

compatibility with the latest machine learning frameworks, imaging libraries, and security updates.

Another assumption is that the system may be used by a wide range of individuals, from those with extensive expertise in radiology and chest X-ray interpretation to those with little prior knowledge. This means the user interface, documentation, and usability features should be designed to accommodate varying levels of experience. Experts may use the system to validate results or support diagnostic tasks, while less experienced users may rely on it as a learning or decision-support tool.

Future modifications are not expected to be strictly necessary in the short term, given the current landscape of medical discoveries and available chest X-ray datasets. However, ongoing updates may be applied to improve performance, efficiency, and accuracy over time. Such updates could involve refining the active learning strategy, enhancing data preprocessing techniques, or optimizing the model architecture. In the long run, major modifications will likely only become necessary if significant new medical insights are discovered or if substantially more comprehensive datasets are released. In that case, the system may be adapted to incorporate new diagnostic markers, expanded categories of chest conditions, or even different types of medical imaging beyond chest X-rays. This flexibility ensures that while the system is stable and effective in its current form, it remains adaptable to future advancements.

General Constraints

The constraints imposed on the system are intended to ensure it remains manageable and always runs effectively. The environment in which the system operates should be based on reasonably modern architecture and hardware. While top-of-the-line hardware is not required, the system must be capable of processing thousands of images and providing adequate storage capacity. If the hardware cannot handle image processing efficiently, system throughput may decrease, and execution errors could occur.

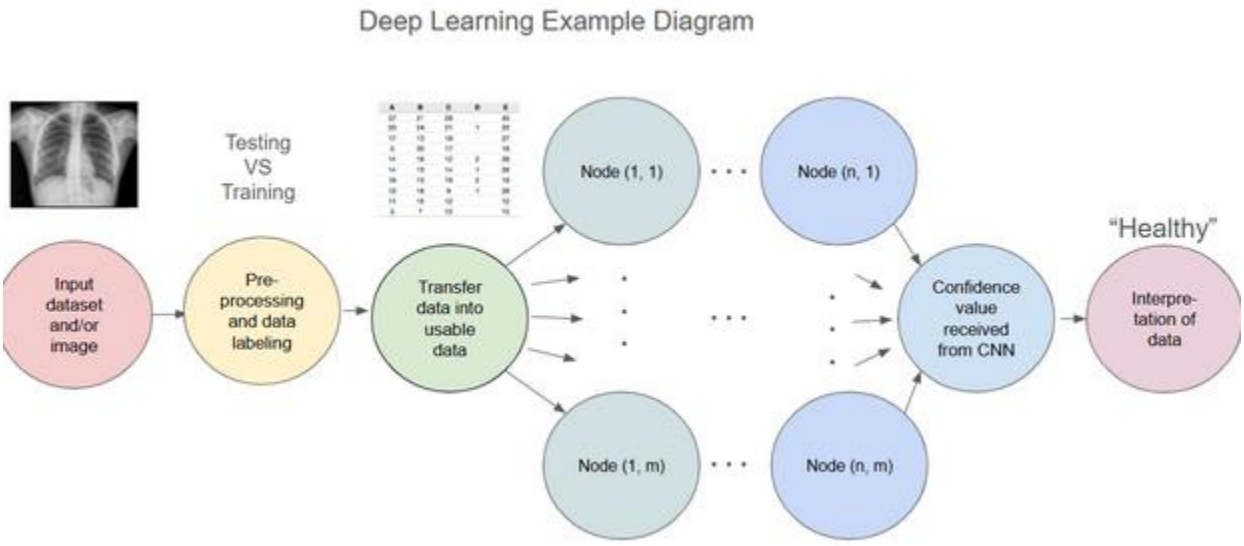
The system is also constrained to use only publicly available datasets related to chest X-rays. It should not process any images that violate privacy or safety laws. Use is limited to chest X-rays and potentially other similar medical imaging tasks, but not in ways that interfere with institutions or businesses in the health and safety field. This active learning system is intended for baseline, routine use in identifying diseases and injuries observable in X-rays.

Although the datasets used are publicly available, meaning security is not the highest priority, basic measures should still be in place to ensure models and the system itself run efficiently and without data leaks. Image labels will be stored securely to prevent tampering. Overall, the system will not require extensive, enterprise-level security but will maintain essential protections for data and images.

Since the datasets will be acquired from online sources and stored locally, the system will not require significant network bandwidth once the data has been downloaded. However, the initial download process may involve large file transfers, as chest X-ray datasets often consist of tens of thousands of images, requiring both time and stable connectivity. After acquisition, all image data will be managed and accessed locally, reducing dependency on constant internet access. This ensures that the system can continue to function effectively even in environments with limited or unreliable network availability.

By operating primarily in a local environment, the system also reduces risks associated with transmitting sensitive medical data over networks, even if the datasets themselves are publicly available. Local storage further allows for faster retrieval and processing of images during active learning cycles, improving system responsiveness. Updates to the dataset or model can still be applied periodically through controlled online access, but day-to-day functionality will remain self-contained. In this way, the system balances the efficiency of local processing with the flexibility of occasional online integration.

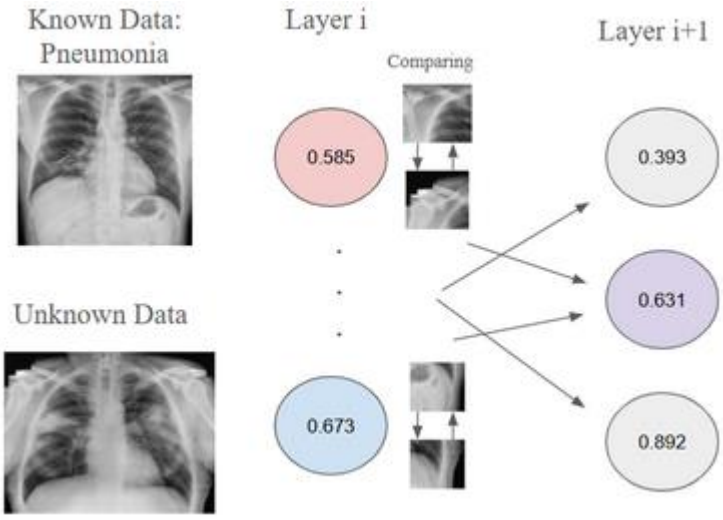
Development Methods



For this project, we must integrate the idea of using deep learning and Convolutional Neural Networks (CNNs) to create an AI that can understand the X-ray images it has been given. In the first section, we will need to implement a way for the program to accept the image (such as a PNG) and be able to implement a way for the model to take data out of said image.

After implementing a method for storing the different images in the tables, we must create two datasets: one for known data and one for test data. We will store most of the images as known data in the table for the model to compare information to. We will use data inside the images, such as the grey-scale rating, for each pixel. We store the data associated with the “known” images on a table that includes the patient's diagnosis, such as the diseases they may have, and the test data in a separate table without diagnostic information.

Once the table is created, the model will then start to compare the unknown data individually with each section in the known section. This is the process that happens during the CNN part of the AI. A basic understanding of a CNN is that the CNN will break the work down into different layers, known as nodes. In each node, a value will be stored, such as an integer or a float value. In the first layer, the model compares a singular pixel in the unknown data to similar pixels in known data and receives a value on its similarity. It will then continue with this process with each pixel in the unknown data until it creates m nodes, where m is the number of pixels. The next layer will then take a grouping of pixels to compare known and unknown data. It will compare and create a new value for each node in the second layer. This process will then be repeated, creating new groupings and nodes, n number of times.



Once the final layer is created using all previous data, the model will come to a final value, which will show us its final verdict and confidence levels. We will then have the program interpret the data, printing out the result for us to see. We can then fix any confidence levels if needed or tweak some of the calculations to make the program more accurate.

Architectural Strategies

The system is structured as a set of modular components, including data ingestion, storage, preprocessing, model training, active learning, and the user interface. Data flows from raw images and metadata through preprocessing and baseline training into the active learning loop, which generates evaluation reports and stores model artifacts.

Python is chosen as the primary language because of its strong support for PyTorch and related libraries. While TensorFlow remains a possible alternative, PyTorch's flexibility and wide community adoption make it better suited for experimentation and integration. The model leverages pre-training to reduce computational costs and the need for large, labeled datasets. Images are stored in organized file systems, while metadata is managed in lightweight formats such as SQL databases, balancing scalability with simplicity.

The active learning loop relies on pool-based sampling that combines uncertainty and filtering diversity, with random sampling maintained as a baseline and ensemble methods recognized as more accurate but less efficient. The system also incorporates validation checks to ensure data

integrity and robust error handling. Future extensions may include semi-supervised learning and explainability features to support potential clinical applications.

System Architecture

The architecture of the Active Learning System is designed to partition responsibilities into well-defined subsystems that would collectively enable efficient dataset management, model training, active learning iteration, and performance evaluation. At a high level, the system is composed of five components: Dataset management, Model Training, Active Learning Engine, Evaluation and Visualization, and User Interface Layer.

The Dataset Management subsystem is responsible for various tasks, like loading the NIH Chest X-Ray dataset and preprocessing images and metadata. This ensures that consistent, model-ready data is supplied to downstream modules. The Model Training subsystem handles both the baseline model initialization and the retraining process that occurs after each active learning iteration.

The Active Learning Engine serves as the core driver of the system. It applies to acquisition strategies such as uncertainty sampling to identify the most informative unlabeled samples for annotation. These selections are then routed back to the dataset management, where labels are added to the dataset.

The Evaluation and Visualization subsystem provides mechanisms for tracking performance after each iteration. It generates metrics like accuracy and recalls while also visualizing learning curves that show how performance improves as more samples are labeled.

Finally, the User Interface Layer ties all components together by providing researchers with an accessible entry point into the system. Through the interface, users can load datasets, begin training, configure active learning loops, and load results.

Detailed System Design

Classification

- Dataset Management: Subsystem/module package with helper classes for I/O, metadata parsing, preprocessing, and dataset splits.
- Model Training: Subsystem service with trainer class and checkpointing utilities.
- Active Learning Engine: Subsystem service with pluggable acquisition strategies.
- Evaluation and Visualization: Subsystem analytics module with metric calculations and plotting utilities.
- User Interface Layer: Presentation layer: web UI (React/Flask/etc.) or interactive notebook widgets.

Definition

- Dataset Management: Responsible for reading the NIH ChestX-ray14 dataset, validating integrity, standardizing images (preprocessing), and maintaining labeled and Unlabeled pools. Produces deterministic train/val/test splits and exposes iterable data loaders.

- Model Training: Initializes the baseline CNN (to be trained on an optimal dataset to avoid bias with the chest X-ray dataset), configures loss, learning-rate scheduler, optimizer, and runs epochs over provided data loaders.
- Active Learning Engine: Coordinates select → label → update → retrain cycles. Computes acquisition scores on the unlabeled pool, selects a batch under a labeling budget, fetches labels (simulated from the dataset), promotes the labeled pool, and triggers retraining.
- Evaluation and Visualization: Computes performance metrics. Produces learning curves, threshold sweeps, and per-round comparison reports (TBD).
- User Interface Layer: Provides the ability to select dataset path, view class histograms, launch baseline training, configure the acquisition strategy and batch size, monitor training, review metrics, and export results.

Constraints

- Dataset Management: Must be able to accurately process large amounts of images. Must comply with privacy and safety laws.
- Model Training: The model should only label the images of the dataset and not modify any images. The system should be able to handle processing.
- Active Learning Engine: Should remain efficient even as the dataset scales, with selection algorithms optimized for large pools.
- Evaluation and Visualization: Must present results in an interpretable manner for both experts and non-experts, without overwhelming users.
- User Interface Layer: User should not inherently be able to interact with any of the inner workings of the model and dataset. Must remain lightweight and responsive, even when handling large datasets.

Resources

- Dataset Management:
  -Disk: Image store + metadata CSVs.
  -Libraries: Image I/O (Pillow/OpenCV), table parsing (pandas), numeric (NumPy)
- Model Training:
  -Hardware: Decent GPU/CPU combo
  -Libraries: Pytorch
  -External: Optional experiment tracker (TensorBoard/MLflow)
- Active Learning Engine:
  -Pools: Access to Dataset Management APIs for labeled/unlabeled sets
  -Storage: Per-round ledger (TBD)
- Evaluation and Visualization:
  -Libraries: scikit-learn, matplotlib
  -Data: Access to predictions and targets
- User Interface Layer:
  -Static storage: resulting images/exports for download.
  -Concurrency: Queues for training jobs

Interface/Exports

Interface: load_dataset(), split_data(), get_batch()

Exports: labeled/unlabeled pools, standardized image tensors, and data statistics.
- Model Training:
  Interface: train_model(), evaluate_model(), save_checkpoint()
  Export: trained model weights, training logs, and performance metrics.
- Active Learning Engine:
  Interface: select_samples(strategy, k), update_labels()
  Exports: List of selected images updated label pools, and acquisition logs.
  k would be an integer likely representing batch size of images for labeling per iteration.
  Strategy would be a parameter telling the active learning engine how samples would be chosen.
- Evaluation and Visualization:
  Interface: compute_metrics(), plot_learn_curve()
  Exports: visualization like ROC/PR curves and JSON metrics
- User Interface Layer:
  Interface: GUI buttons, notebook widgets
  Exports: Interactive views (dashboards, tables) and downloadable (PDF, PNG)

Development

Core Components and Functionality

The data processing pipeline is implemented through several Python files and a configuration file:

- indexer.py: This file reads the NIH Chest X-ray metadata (Data_Entry_2017.csv) to construct a structured table (a pandas DataFrame) linking each image to its metadata. It also filters the dataset to include only the frontal chest X-rays to ensure consistency.
- splitter.py: This file partitions the dataset into train, validation, and test sets using a 70%/15%/15% ratio. Splitting is done by using unique patient IDs. This also helps prevent data leakage across sets.
- preprocess.py: This file creates the image transformation pipeline using torchvision.transforms. It handles necessary steps like resizing images to 224x224 pixels, applying random augmentations for training, converting them to PyTorch tensors, and applying optimal ImageNet normalization.
- dataset.py: This module, via the CXRDataset class, serves as the link between the preprocessed data and the training process, integrating with PyTorch's data pipeline. It loads images using the Pillow (Pil) library, applies the defined transformations, and extracts label information.
- smoke.py: This file is a lightweight integration test used to verify that all data processing components (indexing, splitting, loading) function correctly and interact smoothly before initiating model training.
- config.yaml: This file acts as the central hub for defining all key parameters, allowing the entire pipeline to be easily reconfigured without changing source code. It specifies paths, disease categories, model input size, and the dataset split ratios.

Database Connection

This project does not implement a database connection but instead employs a file-based data storage approach with a dataset that is downloaded on the user's system. The dataset serves as the primary data source for model training, validation, and testing.

Set-Up Process

In the GitHub repository, there are a total of six files that are essential to the overall functionality of the system. The files that need to be downloaded to the user's system can be found in the "source code" folder within the repository. It is highly recommended to keep track of where these source code files are stored for easier configuration and troubleshooting.

The next important component is the dataset itself, which can be found on Kaggle at the following link: https://www.kaggle.com/datasets/nih-chest-xrays/data. First-time users will likely need to create a Kaggle account to access and download the dataset. Once downloaded, the dataset will be contained in a folder named "archive." It is recommended to store this folder in the same directory as your source code files. If that is not possible, make sure to record the exact file path for later reference.

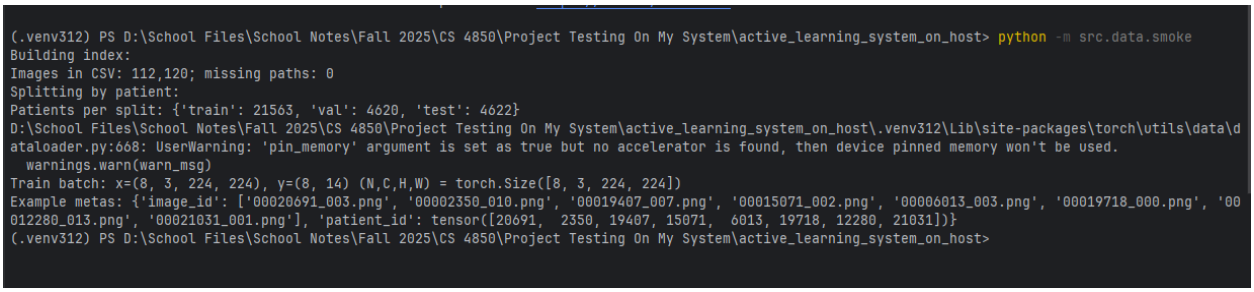
A Python version of 3.10 or later is required to compile and run this system. The latest Python version can be downloaded from https://www.python.org/downloads/. Several dependencies and packages must also be installed for the system to function properly. The primary one is PyTorch, which can be installed by running the command "`pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu124`" in the PowerShell terminal. Further instructions or troubleshooting information can be found on the official PyTorch website at https://pytorch.org/.

Additional required packages include pandas, numpy, and pyarrow, which can be installed using the command "`pip install pandas pyarrow numpy`." Possible packages that will be implemented include tqdm and matplotlib, which can be installed using "`pip install tqdm matplotlib`."

After all dependencies are installed, ensure that the dataset folder (archive) and source files are properly configured and that all file paths match your local directory structure. If the dataset or output folders are stored in non-default locations, update the script variables or configuration entries that reference them, such as those specifying archive/Data_Entry_2017.csv or the output directories for generated files.

To verify that everything is set up correctly, execute the script "smoke.py" by running "`python smoke.py`" in the terminal. This script reads the dataset metadata and generates an indexed output file named index_table.csv, and optionally index_table.parquet, inside the "outputs" directory. Successful creation of these files confirms that the dataset, file paths, and environment are properly configured, and that the system is ready for further training or evaluation.

Here is what the output should look like to help validate successful implementation with all the files.



Challenges

Throughout the development of this classification system, several significant challenges appeared. One of the earliest obstacles involved compatibility issues between the project dependencies and newer versions of Python, which required downgrading from Python 3.13 to 3.12 to ensure essential packages such as PyTorch and torchvision would run correctly once installed. This slowed a lot of initial progress and required several environmental resets and reinstallations. Another major challenge involved optimizing GPU utilization. Although the system initially defaulted to CPU execution, moving training and evaluation onto the GPU required updates to the training loop and DataLoader configuration. Even after enabling GPU usage, performance issues were still present, particularly during evaluation, where processing times reached over 2,000 seconds per epoch for certain subset sizes.

Model performance also presented difficulties. Despite successful training runs, the AUCROC and AUCPR metrics remained near the middle. (~0.50 AUCROC and ~0.05 AUCPR), suggesting that the initial subset sizes, preprocessing pipeline, or training configurations were insufficient. This required adjustments to the batch size, number of workers, subset sampling logic, and normalization settings, along with repeated long training cycles to observe changes. Finally, the complexity of tuning the DataLoader, especially the interactions between the number of workers, memory, persistent workers, and GPU transfer speeds, introduced additional debugging challenges. Understanding which settings genuinely reduced bottlenecks required multiple controlled experiments, each of which would take significant time due to the size of the dataset and model.

These challenges highlight the difficulty of building a high-performance active learning model from scratch, especially one involving large medical-imaging datasets, GPU acceleration, and multi-stage preprocessing. Despite the obstacles, each issue contributed to a clearer understanding of how to optimize the end-to-end system and prepare it for subsequent stages of model refinement.

Test (Plan and Report)

Software Test Plan (STP)

Objectives

- Ensure all components of the active learning pipeline operate reliably.
- Validate correct dataset ingestion and indexing.
- Verify preprocessing, normalization, and DataLoader behavior.
- Confirm GPU detection and utilization within the workstation environment.
- Validate stable execution of the model training loop.
- Ensure model evaluation metrics (AUC, AUCPR, loss) are generated accurately.
- Provide a repeatable procedure to confirm system correctness and robustness.

Scope of Testing

- Dataset ingestion & CSV index validation
- Patient-level dataset splitting
- GPU detection and device assignment
- DataLoader behavior (workers, pin memory, batch loading, non-blocking transfers)
- Training loop execution:
  - Loss reduction
  - Learning rate behavior
  - Backpropagation correctness
  - Gradient clipping behavior
- Evaluation loop execution and metric reporting
- Output file generation (e.g., parquet files, JSON split files, trained model artifacts)

Test Procedures

 Dataset Indexing Tests
- **Verify CSV loading**
- **Validate file path existence**
- **Confirm image assignment to correct patient ID**

 Data Split Tests
- **Validate patient-level uniqueness across splits**
- **Ensure ratio adherence (train/val/test)**
- **Confirm JSON output correctness**

 GPU Utilization Tests
- **Detect GPU availability**
- **Move tensors to GPU**
- **Confirm non-blocking transfers**

 Training Loop Tests
- **Execute training for N epochs**
- **Monitor loss, gradient clipping behavior**
- **Validate optimizer stepping**
- **Confirm model artifacts save successfully**

 Evaluation Loop Tests
- **Run forward pass without gradients**

- **Compute metrics: AUCROC, AUCPR, loss**
- **Generate performance summary**

Software Test Report (STR)

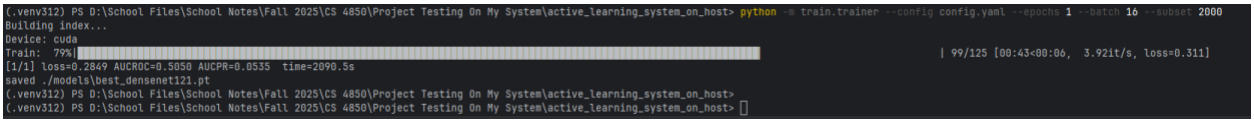
Summary of Tested Components

- Dataset indexing & ingestion
- Patient-level dataset splitting
- GPU detection and utilization
- DataLoader performance and stability
- Training loop execution (subset and full-mode tests)
- Evaluation loop execution & metric reporting
- Output file generation (splits, parquet, trained model weights)

Overall Results

- **Passed**
  - Dataset indexing (paths valid, metadata loaded correctly)
  - Data splitting logic (no patient overlap; valid counts)
  - GPU detection and assignment
  - DataLoader batch loading with workers
  - Training loop execution (loss decreases appropriately)
  - Output file generation (JSON, parquet, model weights)
- **Failed / Needs Optimization**
  - Model AUC performance below the acceptable threshold
  - Evaluation runtime significantly longer than expected
  - Some instability when using high worker counts
  - GPU memory usage suboptimal for large subset runs

Training loop output example showing training progress:



Severity Classification

- **Critical**
  - Prolonged evaluation runtime (affects usability & scheduling)
  - Model performance failing minimum accuracy thresholds
- **Moderate**
  - High GPU memory usage
  - Occasional DataLoader worker timeout under load
- **Minor**

- Training logs could benefit from more detailed reporting
- Metric rounding inconsistencies

Version Control

Our project used GitHub to keep all of our work organized and in one place. The repository stored our source code, documents, and any other files needed for the system and project as a whole. Each team member uploaded their changes to GitHub so everyone could see what was updated and when. This made it easy to keep track of progress and avoid losing work. When someone made a change, it was added as a new version, which allowed us to go back to earlier versions if something went wrong. We also used GitHub's features to note issues, keep track of tasks, and make sure everyone was working on the correct files.

Benefits of Version Control

1. Change Tracking & History

- Every modification to the codebase is recorded.
- Developers can review what changed, when it changed, and who made the change.
- Enables auditing and traceability—critical for debugging and compliance.

2. Collaboration & Parallel Development

- Multiple developers can work simultaneously on different features.
- Version control automatically handles synchronization, merging, and conflict resolution.
- Prevents overwriting or losing work when teams scale.

3. Backup & Recovery

- The repository serves as a complete backup of project history.
- If code is deleted or corrupted locally, it can be restored instantly from the remote repository.
- Protects against accidental loss or system failures.

Directory Organization:

```
active_learning_system_on_host/
│
├── .idea/    # PyCharm / IntelliJ project settings
│
├── .venv312/   # Current active virtual environment (Python 3.12)
│
├── models/              # Saved model weights & checkpoints
│   ├── best_densenet121.pt
│
├── outputs/             # Training logs, CSVs, metrics, exported results
│   ├──index_table.parquet
│   └──splits.json
```

```
├── src/                # Source code for active learning system
│   └──data/            # Dataset classes, transforms, dataloaders
│              ├──__pycache__
│              ├──dataset.py
│              ├──indexer.py
│              ├──preprocess.py
│              ├──smoke.py
│              └──splitter.py
│
│
├── train/              # Training scripts, experiment runners
│   ├── metrics.py
│   ├──model_factor.py
│   └──trainer.py
│
│
└──config.yaml          # Global configuration file (epochs, batch size, paths)
```

Summary

This project demonstrated how active learning and deep learning can be applied to medical imaging using a real-world chest X-ray dataset. By designing a modular, Python-based system that integrates dataset management, CNN-based model training, an active learning engine, evaluation tools, and a user interface layer, we created a complete end-to-end pipeline for iterative learning and performance improvement. The use of publicly available data, local processing, and command-line operation ensured that the system remained accessible, lightweight, and practical for academic use.

Throughout development, we have addressed important challenges, including dependency compatibility, GPU utilization, and performance tuning for large datasets. Although model performance metrics did not yet reach ideal levels, the testing process confirmed that the core components of the system operate reliably and according to design. The structured Software Test Plan and Software Test Report further validated correct dataset ingestion, preprocessing, training stability, and evaluation accuracy, while also identifying areas that require future optimization.

Beyond the technical implementation, this project has provided valuable hands-on experience with real medical imaging data, active learning strategies, and full-system software development practices, including version control and collaborative workflows using GitHub. Overall, the system serves as a strong foundational framework for further refinement, performance tuning, and potential future extensions. Most importantly, it illustrates the practical benefits and real-world challenges of applying active learning to medical image classification.